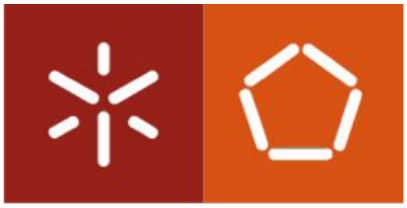


Universidade do Minho
Departamento de Informática
Mestrado Integrado em Engenharia Informática

Análise e Testes de Software: TP2

Diana Lopes , pg38925;
Gonçalo Camaz, a76861 ;
Raul Vilas Boas, a79617

Conteúdo

Capítulo 1

Introdução

O presente relatório foi desenvolvido no âmbito do trabalho prático da UC complementar Análise e Teste de Software do Mestrado Integrado em Engenharia Informática da Universidade do Minho. Neste projeto foi-nos proposto efetuar testes ao software **UMer**, desenvolvido por alunos do 2º ano do nosso curso. Pretende-se avaliar a tanto a funcionalidade do código como das funcionalidades disponíveis aos utilizadores. Desta forma o trabalho realizado assentou sob dois aspetos: Testes unitários e Testes de sistema, que são descritos ao longo do relatório.

Capítulo 2

Projeto: UMer - Testes ao Software

Num *software* desenvolvido nem sempre se consegue garantir o seu correto funcionamento, sem a ocorrência de erros, uma vez que contém no seu código fórmulas e algoritmos complexos.

Uma falha num produto pode ocorrer por diversos motivos, como por exemplo: podem existir requisitos impossíveis de implementar, a especificação pode não estar feita corretamente e também pode haver erros na descrição/implementação dos algoritmos usados.

Visto isto, uma falha pode ser gerada por diversos fatores envolventes no desenvolvimento do software. O teste de software é então um processo que ajuda a determinar a qualidade de um dado produto. Em seguida são apresentados dois dos diversos tipos de testes que se podem realizar para determinar a qualidade da aplicação.

2.1 Alterações realizadas na primeira etapa

Durante a primeira fase do trabalho para testar a funcionalidade de *Login* sentiu-se a necessidade de criar um método na classe UMeR para realizar o login do utilizador no sistema.

```
1  public boolean loginUser(String key, String password)
       {
3
           User user = allUsers().get(key);
5          if (user != null && user.getPassword().equals(password)) {
               this.current_user = key;
7              this.current_class = user.getClass().getSimpleName();
               return true;
9          }
           else {
11             Company company = getCompanies().get(key);
               if (company != null && company.getPassword().equals(password)) {
13                 this.current_user = key;
                   this.current_class = company.getClass().getSimpleName();
15                 return true;
               }
17         }
           if (key.equals("admin") && password.equals("12345")){
19             this.current_user = "admin";
               this.current_class = "admin";
```

```
21                return true;
            }
23          return false;
        }
```

Para além deste método não forma realizador quaisquer tipo de alterações ou acescentos no código do software UMeR.

2.2 Testes Unitários

Os testes Unitários é um tipo de testes de software que se concentra na verificação de cada um dos métodos do projeto de software. São realizados testes para verificar cada método existente nas classes, com o uso de dados suficientes para testar a sua qualidade.

Deste modo, o teste unitário testa o menor dos componentes de um sistema de forma isolada. Cada uma destas unidades define um conjunto de estímulos, dados de entrada e de saída associados aos mesmo estímulos.

Este tipo de testes são de responsabilidade do programador durante a implementação do sistema. Para além disso, este tipo de testes são considerados o estágio inicial da cadeia de testes ao qual um software é submetido. Não testando todas as funcionalidades de uma aplicação, ficando a cargo de outros testes, como a integração ou performance do sistema.

2.2.1 JUnit

O JUnit é uma ferramenta usada para a criação de classes de testes. Cada uma destas classes tem um ou mais métodos para os quais a equipa pretende realizar testes.

Esta framework tem a finalidade de facilitar na criação de casos de teste, além de permitir que os testes criados possam ser reutilizados.

Testes Desenvolvidos

Começou-se por testar se os métodos mais relevantes das principais classes do software *UMeR*, com intuito de verificar se estavam bem implementados. Assim conseguia-se verificar se as diferentes classes estavam vulneráveis a determinadas falhas.

1. Da classe **UMer** foram realizados os seguintes testes:

   - **changeDriverAvailability()**- testa se o estado de um condutor é alterado.
   - **changeDriverVehicle()**- testa se o veículo associado a um dado condutor é trocado.
     O resultado obtido no teste realizado não foi o esperado, detetou-se que após a execução do método `changeDriverVehicle` a informação de "Owner"do veículo não era alterada, apenas era alterada a informação relativa à matricula do carro que o condutor detinha. Posto isto, para resolver este problema, realizou-se as seguintes alterações:

```
    public void changeDriverVehicle(String driver, String
        licencePlate){
2       this.allDrivers.get(driver).setVehicle(licencePlate);
        this.allVehicles.get(licencePlate).setOwner(driver);
4   }
```

   - **registerUser()**- testa se um dado utilizador é registado.
     Foram realizados testes para os diferentes utilizadores do sistema, clientes e condutores. Constatou-se que o método `registerUser` funciona como esperado para registar clientes e condutores que não estão associados a uma empresa. Porém, os condutores associados a uma entidade não estavam a ser registados na aplicação.

Posto isto, para resolver este problema, realizou-se as seguintes alterações:

```
        public boolean registerUser(User u, String company){
        if (this.allDrivers.get(u.getEmail()) == null && this.
            clients.get(u.getEmail()) == null
                && this.companies.get(u.getName()) == null){
            if (u instanceof Client)
                this.clients.put(u.getEmail(), (Client) u.clone());
            else {
                //inicio do codigo alterado
                Driver d = (Driver) u.clone();
                if (company == null)
                    this.allDrivers.put(u.getEmail(), d);
                else if (this.companies.get(company) != null){
                    this.companies.get(company).addDriver(d);
                    this.driversP.put(u.getEmail(), d);}
                else this.driversP.put(u.getEmail(), d);

            }
            return true;
        }
        else return false;
    }
```

- **registerVehicleP()**- testa se um dado veículo é registado.
- **registerCompany()**- testa se uma dada empresa é registada.
- **registerCompanyVehicle()**- testa se um dado veículo é registado corretamente associado à empresa que é indicada.
- **estimatedTime()**- testa se o tempo estimado para uma dada viagem é calculado corretamente.
- **calculateTraffic()**- testa se o tráfico estimado ao longo de uma viagem é calculado de forma coerente para os diferentes veículos.
- **realTime()**- testa se o tempo real de um viagem é calculado de forma correta.
- **addTrip()**- testa se uma viagem é adiciona a cada um dos seus intervenientes (condutor, cliente e veículo) após realizarem a viagem .
- **addRating()**- testa se é atribuída uma classificação ao condutor que realiza a viagem.

2. Da classe **GUI** foram realizados os seguintes testes:

   - **addClient()**- testa se um dado cliente é adicionado corretamente no software pela interface gráfica.
   - **addDriver()**- testa se um dado condutor é adicionado corretamente no software pela interface gráfica.
   - **addCompany()**- testa se uma dada empresa é adicionada corretamente no software pela interface gráfica.

3. Da classe **User** foram realizados os seguintes testes:

   - **addTrip()**- testa se uma viagem é adicionada a qualquer utilizador "tipo"da nossa plataforma.

4. Da classe **Vehicle** foram realizados os seguintes testes:

- **addTrip()** - testa se uma viagem é adicionada a um veículo específico.
- **moneyGeneratedBetween()**- verifica se o cálculo do dinheiro obtido, pelo motorista, nas viagens realizadas está correto.
- **equals()** - verifica se o método **Equals**, da classe Vehicle, está bem implementado.

5. Da classe **Trip** foram realizados os seguintes testes:

   - **distance()**- testa se o calculo da viagem é realizado corretamente.

6. Da classe **Driver** foram realizados os seguintes testes:

   - **addTrip()**- testa se uma viagem é adicionada ao condutor.
   - **addRating()**- testa se a classificação atribuida ao condutor é adicionada com sucesso.

7. Da classe **Car** foram realizados os seguintes testes:

   - **calculateTraffic()**- testa se o tráfico estimado ao longo de uma viagem é calculado de forma correta para o carro.

8. Da classe **Company** foram realizados os seguintes testes:

   - **addDriver()**- testa se um condutor é adicionado corretamente na companhia.
   - **addVehicle()**- testa se um veículo é adicionado corretamente na companhia.
   - **equals()**- verifica se o método **Equals**, da classe Company, está bem implementado.
   - **availiableTaxis()**- testa se o método de verificação de disponibilidade dos taxistas está bem implementado.
   - **pickDriver()**- testa se a seleção de um condutor de uma empresa está a ser realizada de forma correta.
   - **pickVehicle()**- verifica se a seleção do veículo mais próximo está a ser realizada de forma correta.
   - **addTrip()**- testa se uma dada viagem está a ser adicionada à empresa que a realiza.
   - **getDates()**- verifica se as todas das viagens realizadas estão a ser armazenadas.
   - **moneyGeneratedBetween()**- testa se o dinheiro gerado, na realização de viagens, entre duas datas está a ser todo contabilizado.

2.2.2 Cobertura

A cobertura é uma ferramenta utilizada para o calculo de percentagem de código abrangida pelos testes criados. Podendo também ser utilizada para verificar se algum método não está a ser testado.

Utilizar esta ferramenta é importante na medida em que o grupo tem uma melhor perceção da percentagem de código coberta pelos métodos anteriormente explicitados.

Para verificar a cobertura dos testes criados/gerados utilizou-se o *PITest*, sendo este um *puling* do *Maven*. Esta ferramenta executa os testes com um conjunto de de operações (mutações) configuradas, relatando as mutações "mortas"e "sobreviventes". O teste de mutação é um procedimento onde é adicionada uma mutação (erro) ao código, verificando assim os testes que falham se uma mutação for inserida. A eficácia dos testes é medida com a razão entre as mutações que "sobreviveram"e as que "morreram".

Testes Criados

Começou-se por verificar os testes criados anteriormente com a ferramenta *PITest*. De onde se obteve o seguinte resultado que pode ser observado na imagem 2.1.

Name	Line Coverage		Mutation Coverage	
ATS.java	0%	0/25	0%	0/6
ATSLexer.java	0%	0/45	0%	0/12
ATSParser.java	0%	0/884	0%	0/583
Bike.java	0%	0/16	0%	0/10
Car.java	77%	17/22	29%	4/14
CarTest.java	100%	11/11	0%	0/2
Client.java	50%	21/42	31%	4/13
Company.java	85%	75/88	78%	31/40
CompanyTest.java	100%	89/89	39%	12/31
CustomProbabilisticDistribution.java	100%	13/13	13%	1/8
DeviationComparator.java	0%	0/4	0%	0/7
Driver.java	84%	53/63	59%	19/32
DriverTest.java	100%	19/19	60%	9/15
GUI.java	2%	21/929	7%	31/415
GUITest.java	100%	29/29	0%	0/9
Helicopter.java	69%	11/16	10%	1/10
MoneyComparatorC.java	0%	0/4	0%	0/7
MoneyComparatorD.java	0%	0/4	0%	0/7
RatingComparator.java	0%	0/4	0%	0/7
Trip.java	57%	43/75	43%	12/28
TripTest.java	100%	8/8	0%	0/2
UMeR.java	44%	135/310	24%	41/174
UMeRTest_Inicial.java	100%	93/93	21%	5/24
User.java	49%	31/63	65%	11/17
UserTest.java	100%	10/10	33%	2/6
Van.java	0%	0/16	0%	0/10
Vehicle.java	58%	66/113	40%	14/35
VehicleTest.java	100%	25/25	38%	3/8

Figura 2.1: Resultado obtido na execução do PITest nos testes criados.

Testes do EvoSuite

O *EvoSuite* é uma ferramenta que possibilita a geração de testes de forma automática das classes de teste. Cada uma destas classes tem um ou mais métodos para os quais a ferramenta gera testes. Como foi referido numa fase anterior de trabalhos, o código em estudo contém alguns erros. Para uma deteção mais evidente dos erros existentes foram desenvolvidos testes unitários sob os métodos desenvolvidos no projeto.

Com o uso da ferramenta *EvoSuite* obteve-se a geração de 524 testes e uma cobertura de 100% de Line Coverage para a grande parte das classes, imagem 2.2, da aplicação em estudo.

Element	Class, %	Method, %	Line, %
ATS	100% (1/1)	100% (3/3)	28% (7/25)
ATSLexer	0% (0/2)	0% (0/15)	0% (0/45)
ATSParser	0% (0/32)	0% (0/174)	0% (0/884)
Bike	100% (1/1)	100% (4/4)	75% (12/16)
Car	100% (1/1)	100% (5/5)	81% (18/22)
Client	100% (1/1)	100% (15/15)	90% (38/42)
Company	100% (1/1)	100% (24/24)	100% (88/88)
CustomProba...	100% (1/1)	100% (3/3)	100% (13/13)
DeviationCom...	100% (1/1)	100% (1/1)	100% (4/4)
Driver	100% (1/1)	100% (23/23)	100% (63/63)
GUI	100% (1/1)	29% (21/72)	13% (126/929)
Helicopter	100% (1/1)	100% (4/4)	75% (12/16)
MoneyCompar...	100% (1/1)	100% (1/1)	100% (4/4)
MoneyCompar...	100% (1/1)	100% (1/1)	100% (4/4)
RatingCompar...	100% (1/1)	100% (1/1)	100% (4/4)
Trip	100% (1/1)	100% (26/26)	100% (75/75)
UMeR	100% (1/1)	100% (58/58)	87% (270/310)
User	100% (1/1)	95% (22/23)	92% (58/63)
Van	100% (1/1)	100% (4/4)	75% (12/16)
Vehicle	100% (1/1)	100% (31/31)	96% (109/113)

Figura 2.2: Cobertura obtida com os testes gerados na execução do EvoSuite.

Posto isto, correu-se a ferramenta *PITest* de modo a verificar os testes gerados automaticamente. Obteve-se o seguinte resultado que pode ser observado nas imagens 2.3 e 2.4.

Name	Line Coverage		Mutation Coverage	
ATS.java	48%	12/25	0%	0/6
ATSLexer.java	0%	0/45	0%	0/12
ATSParser.java	2%	17/884	0%	0/583
ATS_ESTest.java	100%	7/7	0%	0/2
ATS_ESTest_scaffolding.java	100%	52/52	0%	0/24
Bike.java	100%	16/16	0%	0/10
Bike_ESTest.java	100%	72/72	0%	0/33
Bike_ESTest_scaffolding.java	100%	52/52	0%	0/24
Car.java	100%	22/22	29%	4/14
CarTest.java	100%	11/11	0%	0/2
Car_ESTest.java	100%	83/83	0%	0/34
Car_ESTest_scaffolding.java	100%	55/55	4%	1/25
Client.java	100%	42/42	31%	4/13
Client_ESTest.java	100%	209/209	2%	1/66
Client_ESTest_scaffolding.java	100%	52/52	0%	0/24
Company.java	100%	88/88	78%	31/40
CompanyTest.java	100%	89/89	39%	12/31
Company_ESTest.java	100%	442/442	0%	0/166
Company_ESTest_scaffolding.java	100%	52/52	0%	0/24
CustomProbabilisticDistribution.java	100%	13/13	13%	1/8
CustomProbabilisticDistribution_ESTest.java	100%	17/17	0%	0/5
CustomProbabilisticDistribution_ESTest_scaffolding.java	100%	52/52	0%	0/24
DeviationComparator.java	100%	4/4	100%	7/7
DeviationComparator_ESTest.java	100%	40/40	50%	3/6
DeviationComparator_ESTest_scaffolding.java	100%	55/55	16%	4/25
Driver.java	100%	63/63	59%	19/32
DriverTest.java	100%	19/19	60%	9/15
Driver_ESTest.java	100%	2135/2135	0%	0/1853
Driver_ESTest_scaffolding.java	100%	52/52	0%	0/24
GUI.java	13%	126/929	10%	42/415
GUITest.java	100%	29/29	0%	0/9
GUI_ESTest.java	100%	301/301	2%	1/53
GUI_ESTest_scaffolding.java	100%	55/55	16%	4/25

Figura 2.3: Resultado obtido na execução do PITest nos testes do EvoSuite.

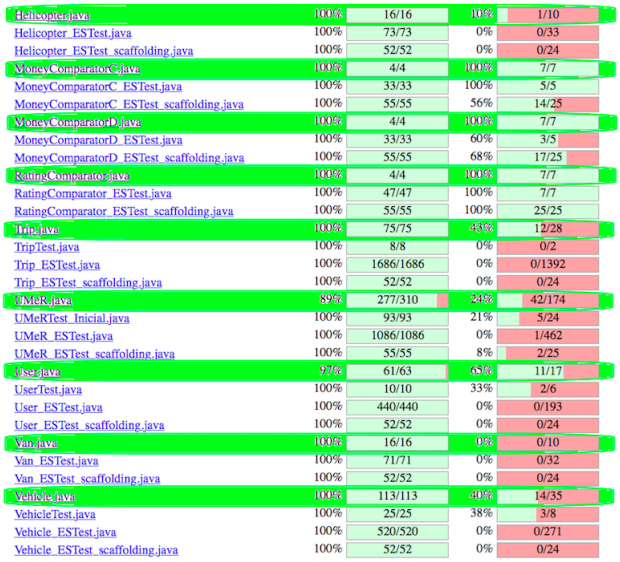
Pela observação dos resultados obtidos verificou-se o seguinte:

- O Line Coverage obtido com os testes criados pelo grupo é quase idêntico ao Line

Coverage obtido com os testes gerados pelo EvoSuite.

- O Mutation Coverage obtido nos testes do EvoSuite em quase nada difere com o Mutation Coverage obtido com os testes descritos anteriormente.

Na generalidade dos casos o facto de ter mais testes concluídos com sucesso e onde se verifica uma Line Covarage de 100% levaria o grupo a concluir que tem garantias, através dos testes, que o software estava bem implementado. Porém, apesar de o EvoSuite gerar muitos testes para testar o software, no caso de ser inserida uma mutação, o resultado (erro) obtido nos testes seria o mesmo que se obteve com os testes anteriormente criados. Como o EvoSuite cria testes de forma aleatória é normal que, por vezes, a criação de testes desnecessários, o que levou a que os mesmos acabassem por ceder às mesmas mutações que os testes iniciais.



Helicopter.java	100%	16/16	10%	1/10
Helicopter_ESTest.java	100%	73/73	0%	0/33
Helicopter_ESTest_scaffolding.java	100%	52/52	0%	0/24
MoneyComparatorC.java	100%	4/4	100%	7/7
MoneyComparatorC_ESTest.java	100%	33/33	100%	5/5
MoneyComparatorC_ESTest_scaffolding.java	100%	55/55	56%	14/25
MoneyComparatorD.java	100%	4/4	100%	7/7
MoneyComparatorD_ESTest.java	100%	33/33	60%	3/5
MoneyComparatorD_ESTest_scaffolding.java	100%	55/55	68%	17/25
RatingComparator.java	100%	4/4	100%	7/7
RatingComparator_ESTest.java	100%	47/47	100%	7/7
RatingComparator_ESTest_scaffolding.java	100%	55/55	100%	25/25
Trip.java	100%	75/75	43%	12/28
TripTest.java	100%	8/8	0%	0/2
Trip_ESTest.java	100%	1686/1686	0%	0/1392
Trip_ESTest_scaffolding.java	100%	52/52	0%	0/24
UMeR.java	89%	277/310	24%	42/174
UMeRTest_Inicial.java	100%	93/93	21%	5/24
UMeR_ESTest.java	100%	1086/1086	0%	1/462
UMeR_ESTest_scaffolding.java	100%	55/55	8%	2/25
User.java	97%	61/63	65%	11/17
UserTest.java	100%	10/10	33%	2/6
User_ESTest.java	100%	440/440	0%	0/193
User_ESTest_scaffolding.java	100%	52/52	0%	0/24
Van.java	100%	16/16	0%	0/10
Van_ESTest.java	100%	71/71	0%	0/32
Van_ESTest_scaffolding.java	100%	52/52	0%	0/24
Vehicle.java	100%	113/113	40%	14/35
VehicleTest.java	100%	25/25	38%	3/8
Vehicle_ESTest.java	100%	520/520	0%	0/271
Vehicle_ESTest_scaffolding.java	100%	52/52	0%	0/24

Figura 2.4: Resultado obtido na execução do PITest nos testes do EvoSuite.

2.3 Testes de Sistema

2.3.1 QuickCheck

Foi também pedido para esta fase que, recorrendo à ferramenta *QuickCheck* gerássemos casos de teste para o programa *UMeR*. Esta ferramenta permite gerar casos de teste aleatórios de acordo com a especificação que foi feita do programa.
No caso em questão, era necessário gerar diversos valores aleatórios tais como:

1. Nomes de Utilizadores (Motoristas, Clientes e Empresas).
2. Veículos (Tipo de veículo, Matrícula, Fiabilidade, Posição).
3. Emails e passwords para cada um dos utilizadores.
4. Métodos do programa *UMeR* relativos aos utilizadores.

Para isso, utilizou-se a linguagem funcional Haskell com o auxílio da ferramenta *QuickCheck*.
Para tornar o processo mais fácil dividiu-se o problema em vários ficheiros diferentes de geradores, isto é, criou-se um gerador para os nomes, um para os emails entre outros. Em seguida são detalhados as diferentes etapas realizadas:

- Com intuito de gerar o nomes definiu-se a função *genNome*.
  Esta função vai buscar um nome da lista de nomes criados no ficheiro *Nomes.hs* e, em seguida verifica se um dado nome existe no estado, isto é possível com o uso da função *notelem*. Caso se verifique que o nome gerado não existe na lista este é adicionado ao estado e é retornado. Assim, caso este mesmo nome apareça novamente não vai ser gerado devido ao facto de já existir no estado.
- Um dos fatores que era necessário ter em conta na criação do *log* era o facto de tanto os emails como as matriculas têm de ser únicas, isto é, não serem geradas repetidas. De modo que tal não aconteça os emails foram criados com base nos nomes dos clientes, assim como o nome do cliente é único o seu email também o será.
- Para as matriculas utilizou-se uma estratégia semelhante em que se gera números únicos permitindo assim que as matriculas nunca serão iguais, no entanto a forma que foi utilizada apenas permite a criação de 100 matriculas.
- Para além destas também se criou funções para gerarem as datas, as coordenadas, as passwords e as empresas.

Na imagem 2.5 consegue-se observar um pouco de como ficou o estado final do gerador de Logs pedido.

Figura 2.5: Estado final do gerador de Logs.

Após definir todos os geradores necessários para verificar as diferentes funcionalidades do software verificou-se, através do comando *ghc Main.hs -e "main»log.txt*, a geração do ficheiro .txt pretendido. Este ficheiro obtido através do gerador definido em haskell pode ser observado na imagem 2.6.

```
registar cliente "RossanaAndrade@hotmail.com" "Rossana Andrade" "6lJkfojb" "Rua Joaquim da
Matosa, 1575, 2825-343 Costa de Caparica" 1998-6-1 (39.0, 80.7) ;
registar condutor "HerveEsteves@gmail.com" "Herve Esteves" "j4DMCqw8" "Rua Joaquim Simoes
da Hora, 700, 2865-513 Fernao Ferro" 1994-1-1 76 ;
registar condutor "DerocilaGoncalves@hotmail.com" "Derocila Goncalves" "Q04b5IlG" "Avenida
Comendador Costa Carvalho, 1395, 3420-428 Tabua" 2004-12-8 "Fast N Furious"  ;
registar empresa "Wings" "zDxeKFpb" ;
registar helicoptero "37-SG-70" 53 "HerveEsteves@gmail.com"  ;
Login "IzalinoLoureiro@sapo.pt"  "pdF17Gi1" ;
Solicitar (7.6, 72.6);
logout ;

Login "HerveEsteves@gmail.com"  "j4DMCqw8" ;
viajar
logout ;


registar cliente "BrunoNogueira@gmail.com" "Bruno Nogueira" "4OQf8M41" "Rua Guilherme
Salgado, 149, 2750-251 Cascais" 1996-7-3 (27.4, 18.1) ;
registar condutor "OlavoNeves@outlook.com" "Olavo Neves" "M5KpZe9h" "Rua Dona Catarina
Figueiredo, 670, 5000-523 Vila Real" 1998-2-15 39 ;
registar condutor "FranclimHenriques@hotmail.com" "Franclim Henriques" "5JlKLsOA"
"Travessa da Igreja, 2309, 4770-361 Mouquim" 1992-3-29 "Wings"  ;
registar empresa "GeTit" "3yPMRTHZ" ;
registar helicoptero "06-CA-15" 60 "AldemarMarques@mail.google"  ;
Login "IzalinoLoureiro@sapo.pt"  "pdF17Gi1" ;
Solicitar (47.3, 8.1);
logout ;
```

Figura 2.6: Estado final do ficheiro log gerado.

2.4 Trabalho Futuro

Futuramente existem alguns pontos que podem ser melhorado no gerador de testes aleatórios. Um aspetos que poderiam ser melhorados é a maneira como o ficheiro *log* é gerado, isto é, em vez de dividir num estrutura em que primeiro se gera os registos e depois as ações fazer de forma a ser mais aleatório. Para além disto, outro ponto possível de ser melhorado é, tanto no programa de geração de testes aleatórios como na gramática, incluir a opção de realizar viagens específicas, ou seja, um cliente poder escolher ou um motorista específico ou o mais próximo. No caso do programa de geração de testes basta ir buscar um email ao *stLogMotoristas* ou ao *stLogEmpresas* garantindo ainda que estes tem veículos e motoristas atribuídos.
É necessário também rever algumas situações pontuais relativas às datas, por exemplo, 28/29 de fevereiro e definir quais os meses que tem 30 e 31 dias, visto que, ao correr o ficheiro de *log* gerado com a gramática definida pelo grupo, foram obtidos alguns erros.